Parallel algorithms for certain problems on DAGs and line graphs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

R SELVAKUMAR

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY , KANPUR

May, 1992

CERTIFICATE

It is certified that the work contained in the thesis entitled *Parallel algorithms for certain problems on DAGs and line graphs* by *R Selvakumar* has been carried out under our supervision and this work has not been submitted elsewhere for a degree.

R.K. Ghosh
Assistant Professor
Dept.of Computer Science
and Engg.
I.I.T. Kanpur

P.Gupta
Assistant Professor
Dept. of Computer Science
and Engg.
I.I.T. Kanpur

May, 1992

ACKNOWLEDGEMENTS

I am extremely grateful to Dr.R.K.Ghosh and Dr.P.Gupta for introducing me to the topic of parallel computation and for the help , encouragement and constant guidance provided by them during the course of this project.

I would also like to express my gratitude to Mr.T.Sokkalingam P. for giving me his valuable time for discussion and for the useful suggestions he gave from time to time.

I thank all my frieds who made my stay at IIT-K a memorable one.

R. Selvakumar

ABSTRACT

Most significant improvements in algorithm performance are achieved by parallelism. Large scale parallelism at the inter-processor level are being used to get fast and efficient parallel algorithm. In order to exploit the potential of the parallel machines one needs to design parallel algorithm for the existing problems.

In this thesis , attempt has been to device parallel algorithm for some graph theoretic problems. The first two problems are on the directed acyclic network and the third problem is on undirected graphs. A parallel algorithm has been designed to evaluated a set of algebraic expressions. This algorithm reduces the number of processors required for the evaluation of the expression. This algorithm takes $O(\log^2 n)$ time with $O(n)$ processors where n is the number of unique computations. The second problem is the computation of the shortest/longest distance on dags. Two algorithms were designed. The algorithm designed for the general problem works with $O(ne)$ processors but the time bound of the algorithm is not sub-logarithmic. The performance of the algorithm on a typical graph is closer to $O(\log^2 n)$. The second algorithm is for a class of directed acyclic graphs computes shortest and longest paths in $O(\log^2)$ time with $O(ne)$ processors. Both the algorithms make use of the modified tree merge technique and they require CREW PRAM model for the computation. Final part of the thesis work deals with parallel algorithm for recognising line graphs. An efficient algorithm has been designed for the PRAM model. An algorithm was designed with $O(n^2)$ processors and $O(\log^2 n)$ time.

Contents

List of Figures

Chapter 1

Introduction

1.1 Parallel computing

A myriad of tasks require fast computation either to get real time response or to process large volume of data set. Parallel systems comprising of a multiple number of processors can provide the support essential to meet the computational requirements. The kind of parallel algorithm designed for a computational problem depends heavily on the nature of the problem and the architecture used to compute the solutions. One obvious way to devise a parallel algorithm is to exploit any inherent parallelism in an existing sequential algorithm. However blindly transforming a sequential algorithm to parallel form is not an effective solution always. Some sequential algorithms have no obvious parallelization. Parallel algorithms made from such sequential algorithms will exhibit poor speed up. These problems are not really inherently sequential. Parallel algorithms for these problems can be developed in two different ways: one can invent a new parallel algorithm or one can adopt another parallel algorithm that solves a similar problem. If the sequential algorithm is not particularly parallelizable, then one must be able to apply some external knowledge of the

problem in order to break the problem into computational tasks which can be executed independently

The performance of an algorithm can be drastically different on different architectures. This is primarily due to communication or synchronization overheads. One should not ignore the communication costs or synchronization costs in determining the complexity of a parallel algorithm. At times, non-computational cost can be higher than the computational cost. In other words, more time is spent routing data among the processors or synchronizing the processors rather than performing the required computation.

In the design of parallel algorithms, the parallel time can be reduced by employing more processors. A constant amount of increase in the number of processors can reduce the parallel time only by a constant factor. If one attempts to reduce the parallel time by a more than a constant factor , then the number of processors employed for the task must be more than a constant factor. In other words , the change in the number of processors employed must be a function of input size. Although parallel algorithms can be designed with any number of processors , the parallel algorithms requiring more than polynomial number of processors are not preferred. The class of problem which can be solved in the polylog time with polynomially many processors is called Nick's class. This class represents the sequential *ptime* algorithms which admit fast parallel algorithms with reasonable number of processors.

1.2 Models of computation

Computer architectures can be classified by the concept of instruction stream and data stream. Depending upon the number of instruction and data streams used by the system, it can be classified into any one of the following four models:

- Single Instruction stream Single Data stream (SISD)
- Single Instruction stream Multiple Data stream (SIMD)
- Multiple Instruction stream Single Data stream (MISD)
- Multiple Instruction stream Multiple Data stream (MIMD)

SIMD and MIMD are the most widely used models of parallel computation. An SIMD computer consists of many identical processors. Each of these processors possess its own local memory where it can store data. All the processors operate under the control of a single instruction stream issued by a central control unit. Equivalently, processors may be assumed to hold identical copies of a single program. Each processor's copy being stored in its local memory. There are many data streams and each processor handles one data stream.

The processors operate synchronously: at each step, all processors execute the same instruction, each on a different datum. At times, it may be necessary to have only a subset of the processors to execute the instruction. This information can be encoded in the instruction itself thereby telling a processor whether it should be active or inactive. There is a mechanism, such as a global clock, that ensures lock-step operation. Thus processors that are inactive during an instruction or those that complete execution of the instruction before others may stay idle until the next instruction is issued. The inter-process communication can be either via a shared memory or an inter-connection network.

This class is also known in the literature as parallel random access machine(PRAM) model. Processors share a common memory in the way a group of people use a bulletin board for communication. When two processors wish to communicate, they do so through the shared memory. The basic model allows all processors to gain access to the shared memory simultaneously if the memory locations are different. SIMD SM can be further divided into the following four subclasses:

- Exclusive Read Exclusive Write PRAM (EREW)
- Concurrent Read Exclusive Write PRAM (CREW)
- Exclusive Read Concurrent Write PRAM (ERCW)
- Concurrent Read Concurrent Write PRAM (CRCW)

EREW PRAM Model

Access to memory locations is exclusive in this model. In other words, no two processors are allowed to simultaneously read from or write into the same memory locations. It is the simples shared memory model. Other shared memory models can be simulated on this model.

CREW PRAM Model

Multiple processors can simultaneously read from the same memory location but no two processors are allowed to write into the same memory location simultaneously.

ERCW PRAM Model

Multiple processors are allowed to write into the same memory locations but the read access remains exclusive , i.e., simultaneous read from a memory location is not permissible.

CRCW PRAM Model

Both read and write privileges are granted. Two processors are allowed to read from or write into same memory location. There is no need to resolve the concurrent read operation. If the two or more processors write in to the same memory location. The write conflict is resolved. This model can be classified based on the method followed to resolve the write conflicts.

1.3 Algorithm performance

The performance of a parallel algorithm performance is judged by its running time, the number of processors used and the cost.

RUNNING TIME

Since speeding up computation appears to be the main reason behind our interest in building parallel computers, the most important measure in evaluating a parallel algorithm is its running time. Running time of an algorithm is the time elapsed from the moment the algorithm starts to the moment it terminates. Theoretical analysis of the running time is done by counting the number of basic operations/steps executed by the algorithm in the worst-case. This yields an expression describing the number of such steps as a function of input and output size. A good indication of the quality of a parallel algorithm is the speed up it produces.

SPEEDUP = worst-case seq. time/worst-case parallel time.

Thus , more the speedup better the algorithm.

NUMBER OF PROCESSORS

The second most important criterion in evaluating a parallel algorithm is the number of processors required to solve the problem. Larger the number of processors an algorithm uses

to solve a problem, the more expensive the solution becomes to obtain. The computational time of a problem can be reduced by employing more processors , as long as there is scope to employ more processors in the computational task.

COST

The cost of a parallel algorithm is defined as the product of the previous two measures.

Cost = parallel running time * number of processors.

Cost equals the number of steps executed collectively by all the processors in solving a problem in the worst-case. If the cost of the parallel algorithm for the problem matches this lower bound , within a constant multiplicative factor, then the algorithm is said to be cost optimal. When no optimal algorithm is known for solving a problem, the efficiency of a parallel algorithm for that problem is used to evaluate the cost. This is defined as follows:

EFFICIENCY = worst-case seq. time of fastest algorithm/ cost of parallel algorithm

1.4 Measures of complexity

Sequential models of computation are equivalent within a polynomial amount of time. As a consequence computability is insensitive to the choice of the model.

A similar claim holds for parallel models of computation. All universal models of expanding parallelism can be shown to be polynomial-time equivalent. Each model can be simulated by the others with at most a polynomial loss of time. In particular, CRCW, CREW and ERCW can be simulated using EREW model in O($log n$) time. Nick's class is robust. The class of the problem remains the same in all the shared memory models.

Although the PRAM model ignores many important aspects of real parallel machines, the essential attributes of a parallel algorithm tend to transcend the modes for which they are designed. If one PRAM algorithm outperforms another PRAM algorithm, the relative

performance is not likely to change substantially when both algorithms are adapted to run on a real computer.

The running time of a parallel algorithm depends on the number of processors executing the algorithm as well as the size of the problem input. Generally, therefore, we must discuss both time and processor count when analyzing PRAM algorithms; Typically, there is a trade-off between the number of processors used by an algorithm and its running time.

1.5 Thesis outline

Chapter 2 gives an outline of the methods to evaluate sets of algebraic expressions. Sequential algorithm for the evaluation of an arithmetic expression take $O(n)$. An optimal parallel algorithm is available for the problem. It needs $O(\log n)$ time. Our attempt is to design a parallel algorithm which can evaluate a set of expression with less number of processors. It eliminates the duplicate computation in the case of common sub-expressions among the various expression of the set.

Chapter 3 deals with the problem of critical path identification on *aoe* networks. Though this chapter deals with the problem of critical path computation, result of this chapter are relevant for shortest and longest path computation on *dag* networks. This problem takes $O(n+e)$ time on a sequential machine. There are known parallel algorithms which take $O(logn)$ time with $O(n^3)$ processors. Attempted was made to reduce the number of processors required for this computation. The method devised assigns one processor for each computation in the *dag* structure used for the computation of the set of expressions

Chapter 4 deals with design of parallel algorithm for recognizing line graphs and to construct the root graph from the given line graph. Best known sequential algorithm for this problem takes $O(n+e)$ time. Attempted has been made here to construct a parallel

algorithm which can recognize line graphs and reconstruct the root graph of the line graph. As a first step, a PRAM algorithm was developed with minimal number of processors. This algorithm takes $O(logn)$ time to construct the root graph from the line graph. This requires $O(n^2)$ processors to construct the root graph.

Finally , chapter 5 briefs the conclusions of the thesis.

Chapter 2

Expression dags

2.1 Introduction

Evaluation of arithmetic expressions is one of the well understood methods for solving problems. Sequential algorithm to evaluate arithmetic expressions takes $O(n)$ time where n is the number of binary operations in an expression. Sequential algorithms evaluate the expressions by evaluating the sub-expression in a bottom-up fashion on the parse tree of the expression. Since the number of computations to be done is n , irrespective of the order in which the sub-expressions are evaluated , the sequential algorithm would take $O(n)$ time. In the parse tree of an expression , each node represents a sub-expression to be evaluated and each edge of the tree represents data dependency between two subexpressions in the tree. The data dependency among the subexpressions imply that on any parallel machine the arithmetic expression evaluation would take at least $O(h)$ time where h is the height of the tree. The height of a binary tree of n nodes can vary from $\log n$ to n but it can not be less than $\log n$. This observation that the height of a binary tree of n nodes can not be less than $\log n$ gives an immediate lower bound of $\Omega(\log n)$ for the evaluation of

arithmetic expressions. On a parallel machine , the arithmetic expressions can be evaluated in two different ways. The expressions can be evaluated in parallel either by rewriting the expression in such a way that the height of the expression tree is more than the minimum possible at most by a constant factor or by evaluating the expression as a whole such that the size of the expression reduces by a constant factor after each iteration of the parallel algorithms. Both the method gives the best possible time bound for the parallel evaluation of the expression tree. But the time bound proved in the case of height reduction algorithm is asymptotic.

2.2 Height reduction method

R.P.Brent , Kuck and Maruyama[BK 73] F.P.Preparata and D.E Muller[PM 75] have designed algorithms which are based on the height reduction method. These algorithm tries to rewrite the expressions into an equivalent form where the height of the tree is bounded by $O(\log n)$. In order to rewrite the expression into an equivalent form , the algebraic laws are used. The evaluation is carried out in two stages. In the first stage of the algorithm , the expression to be evaluated is rewritten in such a way that the height of the parse tree is bound by $O(\log n)$. In the second stage, the new expression is evaluated in the bottom-up order. Sub-expressions which are in the same level are evaluated in parallel. Since the bound on the height of the expression tree is achieved only for higher values of n , the time bound of $\log n$ with $O(n)$ processors is achieved only when the number of binary operations in the expression is a sufficiently high value. In other words , the time bound achieved is asymptotic. Algebraic laws are used to reduce the depth of the expression tree. The arithmetic expression are repeatedly rewritten to equivalent expression where the depth is progressively less.

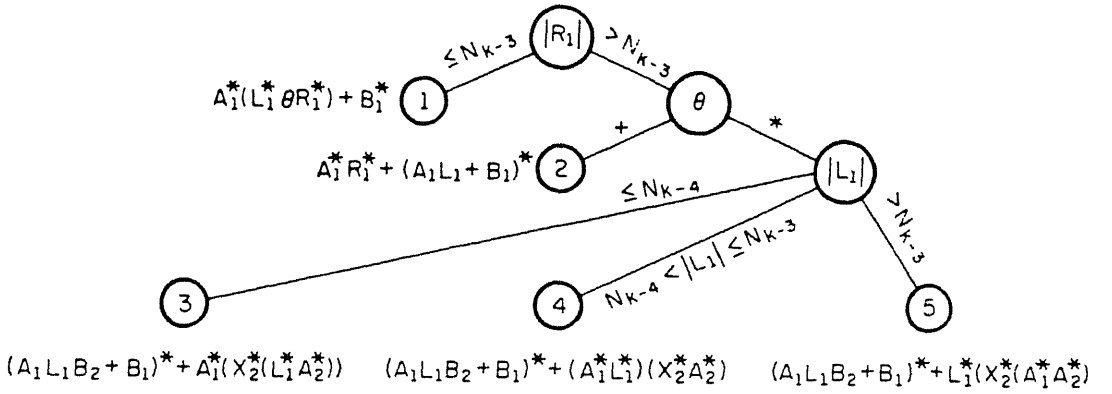


Figure 2.1: Decision Tree

This algorithm recursively restructures the given expression E and returns its equivalent expression E^*, whose depth is bounded by $O(\log n)$. This ensures that the expression can be evaluated faster on a parallel machine. The first part of the algorithm which restructures the expression is a recursive sequential algorithm which takes $O(n^2)$ time to restructure the tree. The second part of the algorithm is the parallel evaluation of the restructured expression. First part of the algorithm is a realisation of the decision tree given in figure 2.1. In each step of the structuring a subexpression X of size N_k is identified in the expression such that the expression can be rewritten into $AX + B$ where the size of A, B, X are such that the over all expression can be evaluated in time $O(k)$. The subexpressions A, B, X are themselves recursively restructured to get the required depth. $\{N_k\}$ in this algorithm represents a sequence of integers satisfying the recurrence relation $N_k = N_{k-1} + N_{k-4} + 1$ with $N_j = j + 1$ for $j = 0, 1, 2,$ and 3.

This algorithm puts an upper bound k on the depth of the expression E, i.e. , $t(E)$ when $| E | \leq N_k$. Since the longest positive root of the recurrence relation is $Z_0 = 1.3802$,

the height of the tree after rewriting will be $t(E) \leq 2.1507 log \mid E \mid + c$. A brief description of the algorithm is given below:

In each step of the algorithm , it finds a subexpression X of the expression E and rewrites it into $E^* = AX + B$ such that $A \leq N_{k-1}, B \leq N_{k-1}, X \leq N_{k-1}$ and $E \leq N_k$. This ensures the convergence of the algorithm after finite number of steps.

Algorithm HT-REDT

STEP 1

Find a subexpression $X_1 = L_1 \theta R_1$ of E , for some operation θ , such that

$\mid X_1 \mid \geq N_{k-1} - N_{k-2} + N_{k-4} + 1$

$\mid L_1 \mid \geq N_{k-1} - N_{k-2} + N_{k-4}$

$\mid R_1 \mid \geq N_{k-1} - N_{k-2} + N_{k-4}$ and

$\mid R_1 \mid \geq \mid L_1 \mid$

STEP 2

If $\mid R_1 \mid \leq N_{k-3}$ set $E^* \Leftarrow A_1^*(L_1^* \theta R_1^*) + B_1^*$ and halt

STEP 3

If θ is $+$ set $E^* \Leftarrow A_1^* R_1^* + (A_1 L_1 + B)^*$ and halt

STEP 4

Find a subexpression $X_2 = L_2 \theta' R_2 of R_1$ for some operation θ' such that

$\mid X_2 \mid \geq N_{k-4} + 1 \mid L_2 \mid \leq N_{k-4} \mid R_2 \mid \leq N_{k-4}$

STEP 5

A. If $\mid L_1 \mid \leq N_{k-4}$ set $E^* \Leftarrow (A_1 L_1 B_2 + B_1)^* + A_1^*(X_2^*(L_1^* A_2^*))$

B. If $N_{k-4} \leq \mid L_1 \mid \leq N_{k-4}$ set $E^* \Leftarrow (A_1 L_1 B_2 + B_1)^* + (A_1^* L_1^*)(X_2^* A_2^*)$

C. If $\mid L_1 \geq N_{k-3} + 1$ set $E^* \Leftarrow (A_1 L1 B_2 + B_1)^* + L_1^*(X_2^*(A_1^* A_2^*))$ Halt.

2.3 Tree contraction method

Miller and Reif[MR 86], Kosaraju and Delcher[KD 88] have proposed tree contraction methods to evaluate any arithmetic expression. Tree contraction methods use the divide and conquer paradigm for the parallel evaluation of an expression. At every iteration of the algorithm the number of nodes which are removed from the tree is a constant factor of the total number of nodes in the tree. Unlike the tree height reduction algorithms, tree contraction algorithms do not try to modify the structure of the expression to be evaluated. These algorithms have many advantages over the height reduction algorithms. Contraction algorithms are easier to implement and their empirical performance is better than the height reduction algorithms. Tree contraction algorithms do not require restructuring of the expression and hence it can be applied for any expression without the sequential pre-processing.

Kosaraju and Delcher's algorithm evaluates the expression in $O(\log n)$ time by employing $O(n)$ processors. The number of processors required for the computation can be reduced by a factor of $\log n$ by applying Brent's[BR 74] theorem for a set of associative operations. This algorithm identifies set of nodes to be removed at every iteration of the algorithm and carries out a set of local operation for each removed node. The outgoing edges and the linear expressions of some of the nodes are modified in such a way that the resulting expression produces the same result as the original tree. At every iteration of the algorithm , the leaf nodes are numbered in the in-order and the even numbered leaf nodes are marked for removal from the expression tree. Each marked leaf node and its parent are removed form the tree. The marked nodes are removed in two steps , left leaf nodes are removed first then the right leaves are removed. This two stage elimination ensures that there is no concurrent read or write in the memory locations accessed by the processors. Parent node of each node is exclusively accessed by that node , hence it can result in concurrent read or

write. However when two consecutive left node or two consecutive right nodes are removed from the tree at the same time , may try to access the same grandparent node. Although the grandparent nodes are same , they access different data for the local computation. Hence , the grandparent access to does not result in concurrent read or write operation. Processor assignment for this algorithm is simple , one processor is assigned for each leaf node in tree and any computation pertaining to that leaf node is done by the processor. Each processor will be active only if the corresponding leaf node is still in the tree. This method gives an optimal algorithm for the expression evaluation on the EREW PRAM model.

2.4 Evaluation of set of expressions

If a set of expressions are to be evaluated at the same time , one need not compute the common subexpressions repeatedly. The computational cost can be reduced by reducing the number of processors employed for algorithm. This can be achieved when each common sub-expression is evaluated just once during the evaluation of the set of expressions. The computations to be done can be represented in the form a directed acyclic graph (*dag*). The parallel algorithm described here gives a way of computing the expressions of the *dag* structure. This algorithm assumes that there is no common sub-expression within an expression. It is shown here that a set of arithmetic expressions with n unique sub-expressions can be evaluated on an CREW PRAM model in $O(\log^2 n)$ time with $O(n)$ processors. This algorithm can be applied for any two operators $\otimes$ and $\oplus$ which are commutative and $\otimes$ is distributive over $\oplus$. Although it is assumed that the operators are commutative the same results can be proved without the commutative property of the operators. For brevity, it is assumed that the expressions consist of just constants , integer addition and integer multiplication. The algorithm can be applied for other arithmetic operators , any operators

which obey multiplicative and additive property.

2.4.1 *DAG* Structure

The set of arithmetic expression to be evaluated are specified in the form of a directed acyclic network. Each node of this network corresponds to a sub-expression to be evaluated. Network contains two types of nodes , internal and terminal. Nodes with out-degree two are internal nodes and nodes with out-degree zero are terminal nodes. Each internal node of the network contains a binary operator while each terminal node of the network contains a constant value. The graph induced by the set of nodes which are reachable from a particular node is a tree and it is the expression tree of the sub-expression corresponding to that node. Each node of the network has memory space to store the value of the sub-expression and each internal node stores the operator which corresponds to that node. Each node is connected to two nodes , left and right nodes , which represent the left and right operands of expression corresponding to the node. Edges in the network specify the interrelation between the sub-expressions in the network. An edge between any two node in the network indicates that the value of the node at the tail of the edge depends on the value of the node at the head of the edge. Each edge of the network stores a linear expression in the form of a pair. It represents the contribution of the left or right sub-result towards the higher level subexpression. The linear expression is called contribution vector. The contribution vector is modified in such a way that the expression *dag* at all point of time represents the same values as the original expressions. All the contribution vectors are initialized to $[1, 0]$ at the start of the evaluation process. This initial value represents all the expression of the set. This initial value depends upon the operator which are used in the expressions. After each iteration of the algorithm , some nodes of the network are removed and the contribution

vector are modified for the remaining edges of the network in such a way that the value of remaining nodes in network is unaffected. One processor is assigned to each of the nodes in the network and the processors will be active whenever a computation is done for the corresponding node in the tree. This algorithm selects a set of non-overlapping nodes by constructing adjacent trees for the *dag* structure instead of the in-order method which is normally used for tree-structured computations.

2.4.2 Evaluation procedure

Each edge in the network stores a linear expression $aX + b$ which is denoted by the pair $[a, b]$. If X is the value of the node at the head of the edge. Value at a node in the network is defined recursively. For any node u with operator $\odot$, if the values at its two sub-expression nodes are VAL_L and VAL_R and the expression at the left and right edge are $(a_l X + b_l)$ and $(a_r X + b_r)$, respectively , then the value at u is defined to be the result of the expression

$$(a_l VAL_L + b_l) \odot (a_r VAL_R + b_r)$$

The expressions at the edges which point to u indicate the contribution of the value at u to the value of the node at the higher levels. At each phase of the algorithm, we ensure that the value at every node of the network defined in this way equals the value of the original sub-expression. To make this true initially, we begin with the expression X., $a_e = 1$ and $b_e = 0$, stored at every edge e of the network.

The algorithm works in two phases. In each step of the first phase , the *dag* is contracted by removing some of the terminal and internal nodes of the *dag* structure and the expressions are partially evaluated. In the second phase of the algorithm the expression are evaluated. Rake and clip are used during the first phase of the algorithm to contract the expression *dag*. Rake operation can be applied to any internal node with at least one terminal node.

Clip can be applied only if both the left and right nodes are terminal nodes.

Rake operation

The rake operation can be applied for any node with one or more terminal nodes. Rake operation removes the concerned internal node and its specified terminal node. Two consecutive rake operations can result in data loss. In order to ensure that correctness of the computation the rake operation is applied only after a marking which avoids overlapping rake operation and generates optimal number of rake operations. To apply the rake operation to an internal node and its terminal node w we have to

1. disconnects w and the internal node from the network and sibling of w becomes the son of parent of the internal node.

2. computes $[a, b]$ for the newly established connection between the parent of the internal node and the sibling of w. $[a, b]$ can be computed by computing the coefficient of X and the constant term.

   if w is a left node $a_{par(w)}((a_w c_w + b_w) \odot_{par(w)} (a_{sib(w)} X + b_{sib(w)})) + b_{par(w)}$

   if w is a right node $a_{par(w)}((a_{sib(w)} X + b_{sib(w)}) \odot_{par(w)} (a_w c_w + b_w)) + b_{par(w)}$

Clip operation

The clip operation is applied only to the internal nodes of the type LR. An internal node becomes a fully computed node as soon as the clip operation applied to it. Any internal node which is subjected to clip operation is not included in the dependency tree construction. Only the partially evaluated nodes generated by the rake operations are included in the dependency tree. Clip operation simply evaluated the node value and stores the result of the computation in the internal node and its left and right nodes are disconnected from the node.

Dependency tree are constructed during the first phase of the algorithm. If an internal node with one leaf node is removed from the *dag* then the evaluation of the internal node can be completed till the value of the of its other operand is known. Each step of contraction operation generates certain dependencies. These are computed in the form of a tree. Since the number of contraction steps executed is bounded by $\log n$, the height of the dependency tree can be greater than $\log n$. The expressions of the *dag* structure are evaluated in the second phase of the algorithm in the order specified by the dependency tree.

Non-overlapping internal nodes are identified by constructing a set of adjacency trees. These trees represent the concurrent data requirement for the removal operations executed at various internal nodes in the *dag* structure. This partitions the set of nodes in the *dag* into many groups where nodes of an adjacency tree forms a group. Nodes of each group are split in to two non-overlapping sets and the set of nodes having at least half of the total number of nodes in the group is marked for removal. Nodes of each such set is removed from the *dag* and the *dag* structure is contracted As a first step in the process of constructing adjacency trees , Each internal node in the network is assigned a label depending upon the terminals connected to it. The possible labels are NONE, LEFT, RIGHT and LR. These labels represent the potential local operations for the internal nodes. Each node in the *dag* is represented by a node in the adjacency trees. if node is LEFT or RIGHT then it is connected to the node of right or left node respectively in the trees and a value 1 is stored in the node. If a node is a NONE or LR node then it is connected to itself and a value 0 is stored in the node. Distance of the nodes from the root of their trees is computed using pointer doubling technique. Nodes of a tree or split into two set , node with even distance from the root node and nodes of odd distance from the root node. For each group , the set with the maximum number nodes is selected and the nodes of this set are eliminated from

the *dag* structure in the next phase of contraction operation. Labels are again assigned for each of the internal nodes in the *dag*. And contraction operation is applied for the LR internal nodes along with their left terminals. The contraction step is repeated till the tree contains only unconnected internal nodes. At this stage the second phase of the algorithm starts. In the second stage of the algorithm the expression are evaluated in the order specified by the dependency tree. The second phase of the algorithm is necessary if one wants to evaluated all the expressions in the set.

Algorithm

PHASE I:

for $i = 0$ to $\log n$ do

1. Set the node type for each of the internal nodes.
2. Apply clip operation to the internal nodes of type LR.
3. Set the node type for modified internal nodes.
4. Mark a set of non-overlapping internal nodes in each tree.
5. rake the marked internal nodes and its terminal node and recompute the linear expressions for the siblings of the terminal nodes which are removed

PHASE II:

for $i = \log n$ down to 0 do

compute expressions of nodes at height i in the dependency tree

To mark a set of non-overlapping internal nodes in each tree one may do the following:

1. Partition the network in to many groups. If two adjacent nodes are LEFT or RIGHT type internal nodes then they are put in the same partition. Nodes of each partition and their edges between them form a in-tree.

2. Compute the distance of each node from the root node of its adjacency tree.

3. Split the nodes a partition in to two groups. Even distance nodes and odd distance nodes are gathered separately.

4. In each partition , the bigger group is marked for the rake operation.

In the first phase of the algorithm , the linear expressions are recalculated after every rake step. This step ensures that the value of subexpression which are present in the *dag* are unaffected through out the computation. In the second phase of the algorithm , the linear expression are used to calculate the exact result of the remaining subexpressions.

CREW model is used for the computation. None of the operations require concurrent write operation. The concurrent read is done in the tree doubling operation. The processor assignment simple in this algorithm. One processor is assigned for each node in the directed acyclic structure. And all the operations pertaining to any node is done by its processors. This imply that the number processors required for the computation is exactly n.

Local operations , Rake and Clip , which are done to contract the *dag* take O(1) time because they need to compute only a fixed amount of take for each application of the operation. The adjacency tree construction can be done in constant amount of time. The process of identifying the nodes for rake operation takes $O(\log n)$ time because this involves computation of distance of the nodes from their root nodes and the sorting of nodes on the basis of root node and distance. The contraction operation is executed $\log n$ times. Hence , the first phase of the algorithm takes $O(\log^2)$ time. The second phase of the algorithm involves computation of expressions. This requires constant amount of time.

Chapter 3

Computation on an dag

3.1 Overview

Computation of shortest and longest path on a directed network is one of the extensively studied graph theoretic problems. Many sequential and parallel algorithms have been designed for this problem. Computation of the shortest or longest distance on a network takes $O(n+e)$ time on a single processor machine. Many parallel algorithms have been designed with polylog parallel time. Dekel et al. [DN 81] proposed a parallel algorithm of $O(\log^2 n)$ time complexity for computing the shortest distance on a non-negatively weighted. This algorithm can be executed on perfect shuffle and cube connected cycle. Chaudhuri and Ghosh[CG 86] have proposed a parallel algorithm which requires $O(\log d \ loglog \ n)$ time bound where d is the diameter of the network (i.e., the number of edges in the longest path from the start node to the terminal node of the network. Chaudhuri has proposed distributed algorithm to analyze the *dag* structure for critical activities. All these algorithms employ $O(n^3)$ processors to compute result. Chaudhuri[CH 90] has presented an adaptive parallel algorithm for analyzing *aoe* networks on an SIMD-SM computer without read or

write conflicts. This algorithm assumes a bounded parallelism. The time bound achieved by the algorithm is $O(n^{1+h})$ with $O(n^{1-h})$ processors , where $h(0 \leq h \leq 1)$ depending upon the number of processors available. The computational cost of these[DN 81, CG 86, CH 90] parallel algorithms are much higher than the cost of the sequential algorithm. The known parallel algorithms employ $O(n^3)$ processors to get polylog time performance. We attempted to reduce the number of processors employed for the directed acyclic graphs. The computations on the directed acyclic graphs have many applications. Topological ordering of events , the analysis of activity-on-edge network are two important application of this problem. Analysis of *aoe* network involves project time calculation and identifying the critical activities. Path construction and walk construction are similar. The only difference is in the occurrence of the nodes. Path can use a node at most once in the entire sequence. On the other hand , walk can have a node any number of times. The absence of the loops in the directed acyclic graph ensures that the path construction is as simple as the walk construction. Hence , the computation of shortest / longest path can be done similar algorithm. and their complexity is same on directed acyclic graphs. We attempted to reduce the computational cost of the parallel algorithm for the path problem on *dags*. The second section of the chapter talks about the topological ordering problem which is a path problem on unit weighted *dags*. This problem is a simpler problem than the shortest or longest problem on the weighted *dags*. The third section of the chapter discusses a known parallel which is based on the tree merge technique. The fourth and final section describes two new algorithm for the path problem.

3.2 Topological ordering

Topological ordering problem is the longest path computation problem the unweighted directed acyclic graphs. Topological ordering of the nodes of a *dag* is a linear ordering of the nodes with the property that if node I is a predecessor of node J in the *dag* then node I precedes node J in the linear ordering. This problem can be solved by the general matrix multication technique. But , the number of processors required for such an implementation will be n^3. The number of processors can be appreciably reduced by using strassen[ST 69] recursive multiplication technique. With this technique the number of processors employed can be reduce to $O(n^{2.38})$. This algorithm performance is asymptotically better than the earlier algorithm.

3.3 Longest path Algorithms

3.3.1 Definitions and Notations

In this section a number of definitions and notations are introduced. The term 'graph' refers to a directed acyclic graph. Given a node x in a graph $\mathcal{G}$, a tree containing all nodes reachable from x through a path consisting of 2^j or less number of arcs , where j is an integer satisfying $0 < j \leq \log n$, is denoted by $T(x,j)$. The parent of a node $z \neq x$ in tree $T(x,j)$, denoted by parent($z \mid T(x,j)$)is a node y if there exists an arc $y \rightarrow z$ in $\mathcal{G}$. The tree $T(x,0)$ contains all nodes y such that $x \rightarrow y$ is an arc in $\mathcal{G}$. Dist($y \mid T(x,j)$) denotes the longest possible distance from node x to y on the graph using at most 2^j edges. Level($y \mid T(x,j)$) denotes the edge distance of the node y from x when at most 2^j edges are used for the traversal. Select($y \mid T(x,j)$) indicates that y is a terminal at j^{th} level and is selected for tree merge.

3.3.2 Known algorithm

The longest path on a *dag* network can be calculated by $\log n$ application of tree merge algorithm. The longest path algorithm takes unit height trees as input which is nothing but the set of outgoing edges for each node in the network. Tree merging algorithm produces $T(x, j+1)$ from each tree $T(x, j)$, for x belonging to a graph $\mathcal{G}$. The tree $T(x, j+1)$ is obtained from the tree $T(x, j)$ and the trees $T(t, j)$ where t is a terminal node in $T(x, j)$. The nodes $\mathcal{G}$ are identified by 1 through n. The set of trees $T(x, j)$, for each x belonging to $\mathcal{G}$, from the input to the algorithm.

Input :

A tree $T(x, j)$, for each node x in $\mathcal{G}$, specified by Parent$(y \mid T(x, j))$ y = 1,2,....n.

Output :

A tree $T(x, j+1)$, for each node x in $\mathcal{G}$, specified by Parent(y|T(x,j+1)) y = 1,2,....n.

step 1 [Compute Distance]

For each pair x, y = 1,2,....n Set distance of node y from root node in the tree of x

step 2 [Identify terminal nodes]

For each pair x, y = 1,2,....n

Identify all terminal nodes t_p , p = 1,2,..r such that the two condition are satisfied

1. Dist$(y \mid T(x, j)) <$ Dist$(y \mid T(t_p, j))$ + Dist$(t_p \mid T(x, j))$

2. Parent$(y \mid T(t_p, j)) \neq 0$ and Parent$(t_p \mid T(x, j)) \neq 0$

For any other terminal node $s \neq t_p$ of $T(x, j)$ such that Parent$(y \mid T(s, j) \neq 0$

Step 3 [Select terminal node]

For each pair x, y = 1,2,....n

Find the node t^* element $t_p \mid p$ = 1,2,....n such that

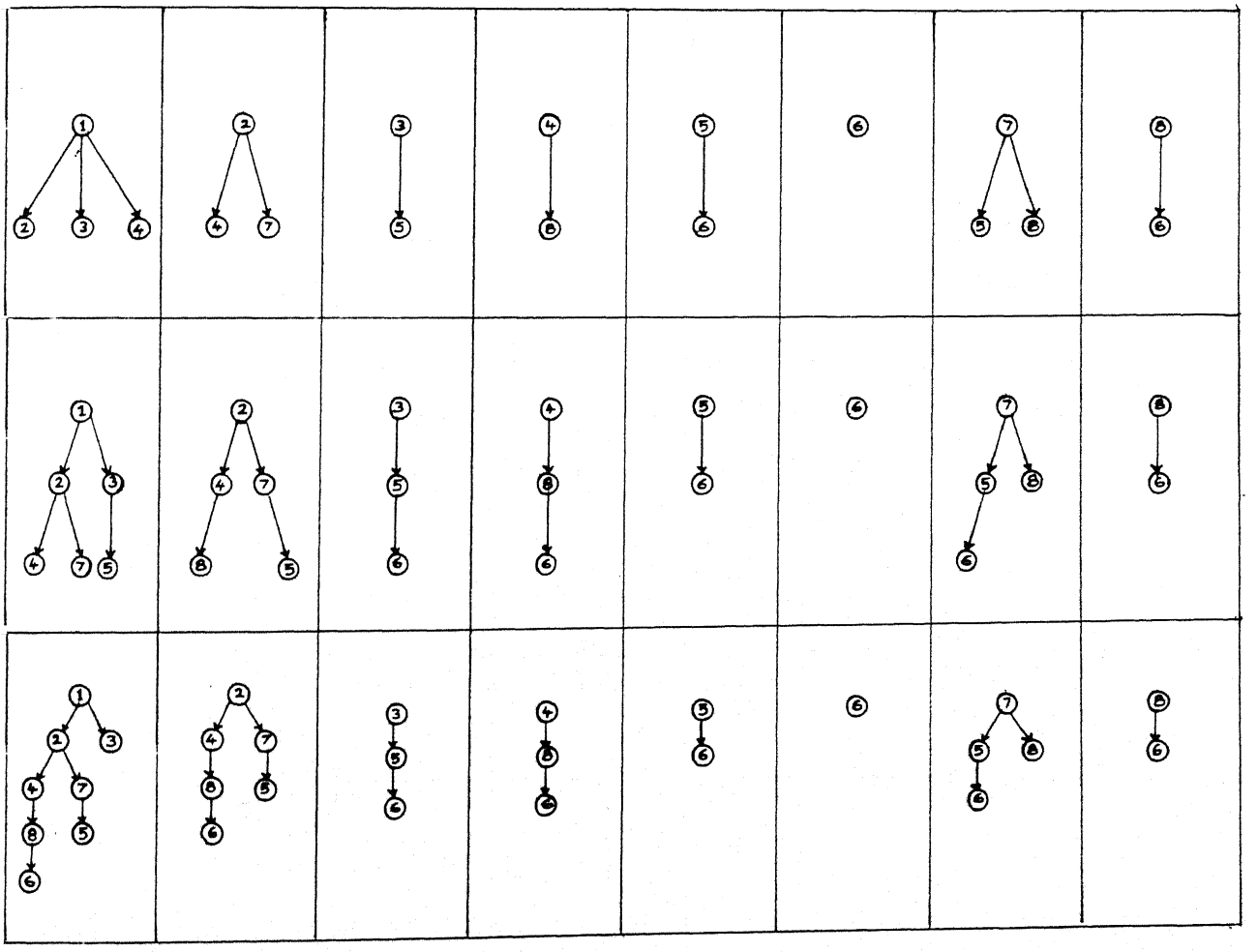
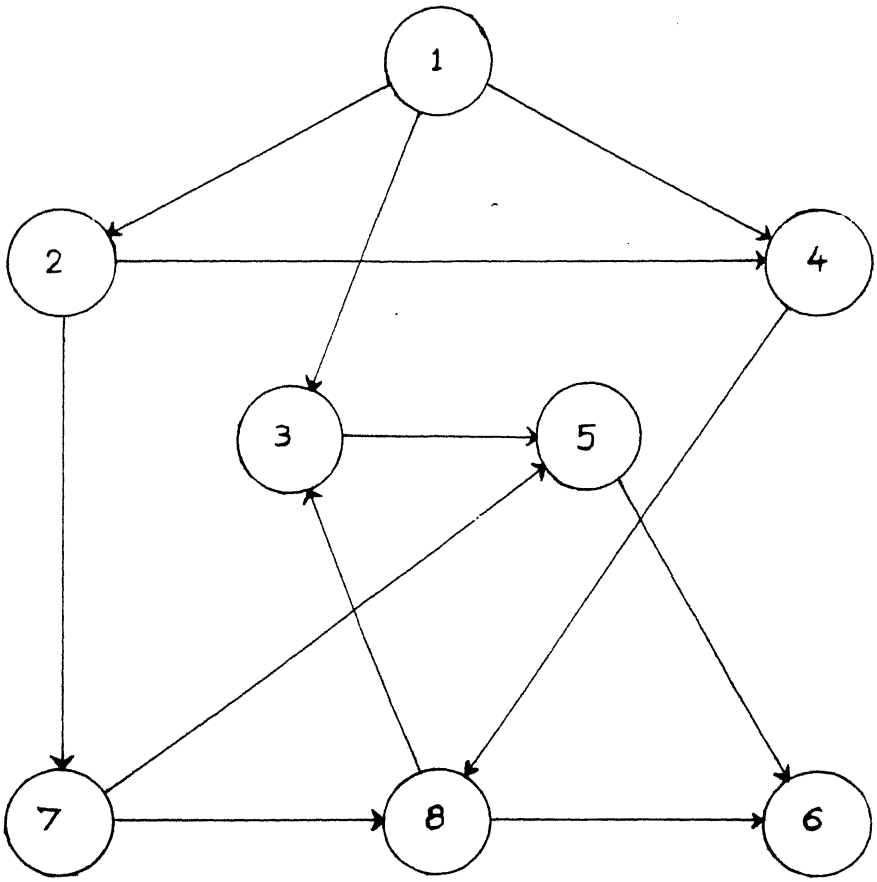


Figure 3.1: An example for tree merge method


1
2
4
3
5
7
8
6

$$\text{Dist}(t^* \mid T(x,j)) + \text{Dist}(y \mid T(t^*,j)) = \text{Max}\ \{\text{Dist}(t_p \mid T(x,j)) + \text{Dist}(y \mid T(t_p,j))\}$$

Step 4 [Set parent node]

For each pair $x, y = 1,2,...n$

Set $\text{Locate}(y \mid T(x,j)) = t^*$

Set $\text{Parent}(y \mid T(x,j+1)) = \text{Parent}(y \mid T(\text{Locate}(y \mid T(x,j)),j))$.

Longest path algorithm

1. $k = 1$

2. Merge k^{th} level trees to produce the $k + 1^{th}$ level

3. $k = k + 1$

4. Repeat steps 2 and 3 until $K = \lceil \log n \rceil$

The first step of the tree merge algorithm takes O($\log n$) to compute with n^2 processors. Implementation of this algorithms is simple. We need to compute the distance of each node from its root node using the the pointer doubling technique. Since the there are n trees and each tree contains at most n nodes the number of processors can not exceed n^2. The second step of the algorithm can be executed in O(1) using n^3 processors. In the third step of the algorithm , the maximum distance is computed for each [node,root] pair. It takes n^3 processors to execute this step. The last step of the algorithm can be executed using n^2 processors in constant time. Hence , the computation of longest path takes O($\log^2 n$) time with n^3 processors.

3.4 New algorithms

3.4.1 Algorithm I

The stifled tree merge method described here attempts to reduce the number of processors employed for the computation of longest and shortest path on a directed acyclic graph. This algorithm employs O(ne) processors. This algorithm assumes PRAM (SM) CREW model for the computation.

Input :

A tree $T(x,j)$, for each node x in $\mathcal{G}$, specified by Parent($y \mid T(x,j)$) y = 1,2,...n.

Output :

A tree $T(x,j+1)$, for each node x in $\mathcal{G}$, specified by Parent($y \mid T(x,j+1)$) y = 1,2,...n.

Step 1 [Prune Tree]

For each pair x,y = 1,,...n If Level($y \mid T(x,j)$) > 2^j the Parent($y \mid T(x,j)$) = 0

step 2 [Compute Distance]

For each pair x,y = 1,2,...n Set distance of node y from root node in the tree of x

Step 3 [Compute Level]

For each pair x,y = 1,2,...n Set level of node y from root node in the tree of x

Step 4 [Select terminals]

For each pair x,y = 1,2,...n sort the set of pairs (x,y) where y is terminal. select one pair (z,y) for each y in the list. and set Select($z \mid T(x,j)$) = TRUE; else Select($z \mid T(x,j)$) = FALSE;

step 5 [Identify terminal nodes]

For each pair x,y = 1,2,...n

Identify all terminal nodes t_p , p =1,2,..r such that the three conditions are satisfied

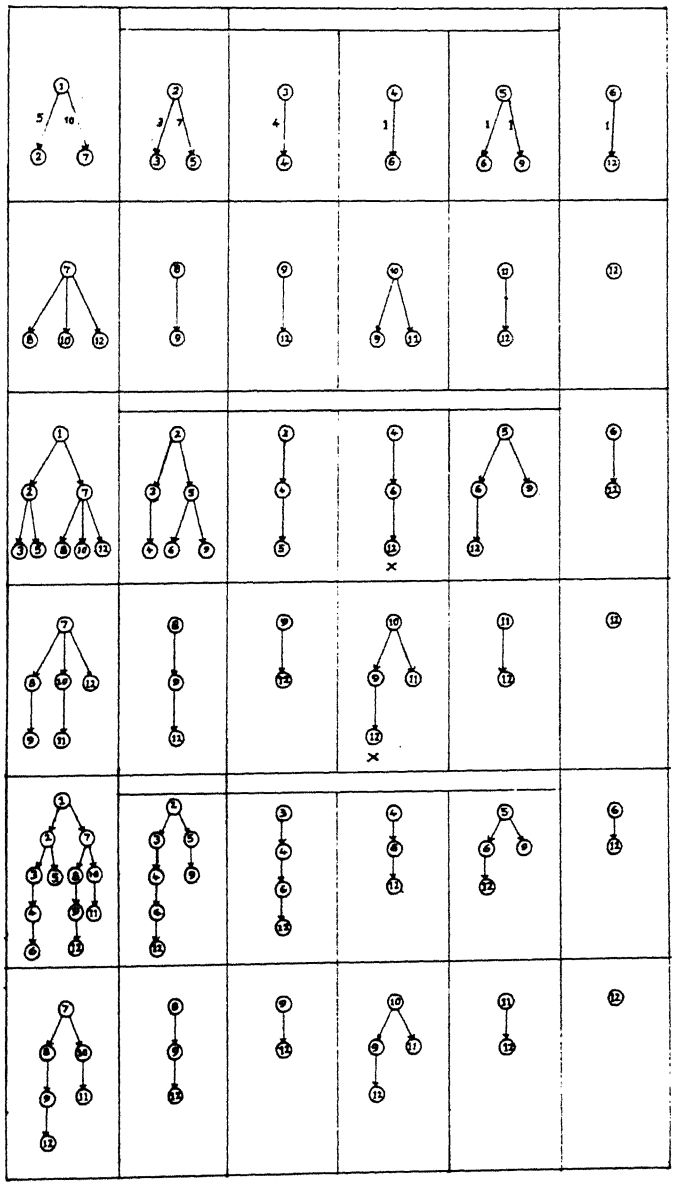
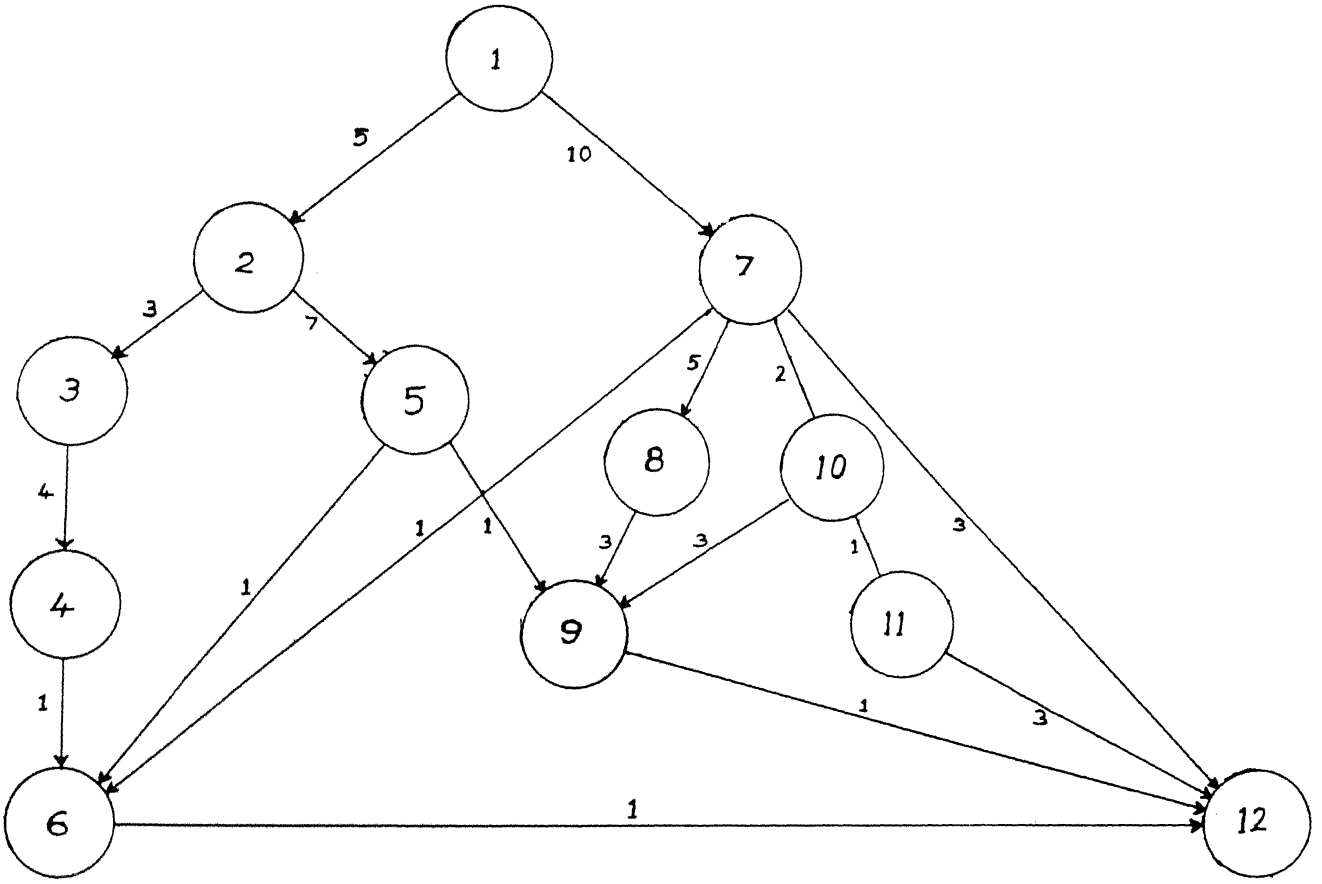


Figure 3.2: An example for stifled tree method


1
5
10
2
7
3
7
3
5
2
5
8
10
4
1
1
3
3
1
4
1
9
11
1
3
1
1
3
6
1
12

1. Dist$(y \mid T(x,j)) <$ Dist$(y \mid T(t_p,j))$ + Dist$(t_p \mid T(x,j))$

2. Parent$(y \mid T(t_p,j)) \neq 0$ and Parent$(t_p \mid T(x,j)) \neq 0$

3. Select$(y \mid T(x,j))$ is TRUE

For any other terminal node $s \neq t_p$ of $T(x,j)$ such that Parent$(y \mid T(s,j)) \neq 0$

Step 6 [Select terminal node]

For each pair x, y = 1,2,...n

Find the node t^* element of $\{t_p \mid p = 1,2,...n\}$ such that

Dist$(t^* \mid T(x,j))$ + Dist$(y \mid T(t^*,j))$ = Max $\{$Dist$(t_p \mid T(x,j))$ + Dist$(y \mid T(t_p,j))\}$

Step 7 [Set parent node]

For each pair x, y = 1,2,...n

Set Locate$(y \mid T(x,j)) = t^*$

Set Parent$(y \mid T(x,j+1))$ = Parent$(y \mid T($Locate$(y \mid T(x,j)),j))$.

Any node which is away from the root of its tree by a distance greater than 2^j is removed from the trees. This done to ensure that the terminal nodes which are used for the construction of trees of height 2^{j+1} are exactly 2^j edges away from the root node with respect to the j^{th} level trees. The distance of level of the nodes are calculated on the pruned tree. Distance and level calculation can be implemented with the pointer doubling technique for the trees. The next step of of the algorithm is to identified a list terminal nodes where the trees will be grown. This step of the algorithm list all possible terminal node is the form of [edge number,tree] pair. These pairs are sorted and e terminal nodes are identified to construct the next level trees. After the terminal nodes are identified for merging , the $n * e$ distances are written in a table of size n by e. $j + 1^{th}$ level maximum distance is is computed for each [tree,node] pair using the $n * e$ sized matrix. For each node

, the parent is changed if the maximum of the $j+1^{th}$ level is greater than the the maximum of the j^{th} level.

The first three steps of the algorithm are computed with the pointer doubling technique. This requires n^2 processor to compute in O(log n) time. The fourth step of the algorithm is to pick out one e terminal nodes out the possible terminal nodes. We can the pick out the e terminal nodes by sorting the pairs and marking one node. for each edge number which is present in the list of terminal nodes. The sorting operation can be done with (n^2) processors and O(log n) time. The terminal node selection can be done in constant time with the same number of processors. longest path for each node can be selected in O(log n) time with ne processors. The longest path tree can be computed by repeated application of stifled tree merge method. This algorithm computes the longest path for the nodes the graph with respect to the root node of the directed acyclic graph. This algorithm gives the longest path tree only with respect to the root node of the directed acyclic graph.

3.4.2 Algorithm II

Reversed tree method

Input :

A tree $T(x,j)$, for each node x in $\mathcal{G}$, specified by Parent($y \mid T(x,j)$) y = 1,2,...n.

Output :

A tree $T(x,j+1)$, for each node x in $\mathcal{G}$, specified by Parent(y|T(x,j+1)) y = 1,2,...n.

step 1 [Construct intermediate graph , $IG(0)$]

For each triplet x,y,z = 1,2,...n if edge $y \rightarrow z$ is present in $T(x,j)$ then enter edge $y \rightarrow z$ in $IG(0)$.

step 2 [Duplicate the intermediate graph]

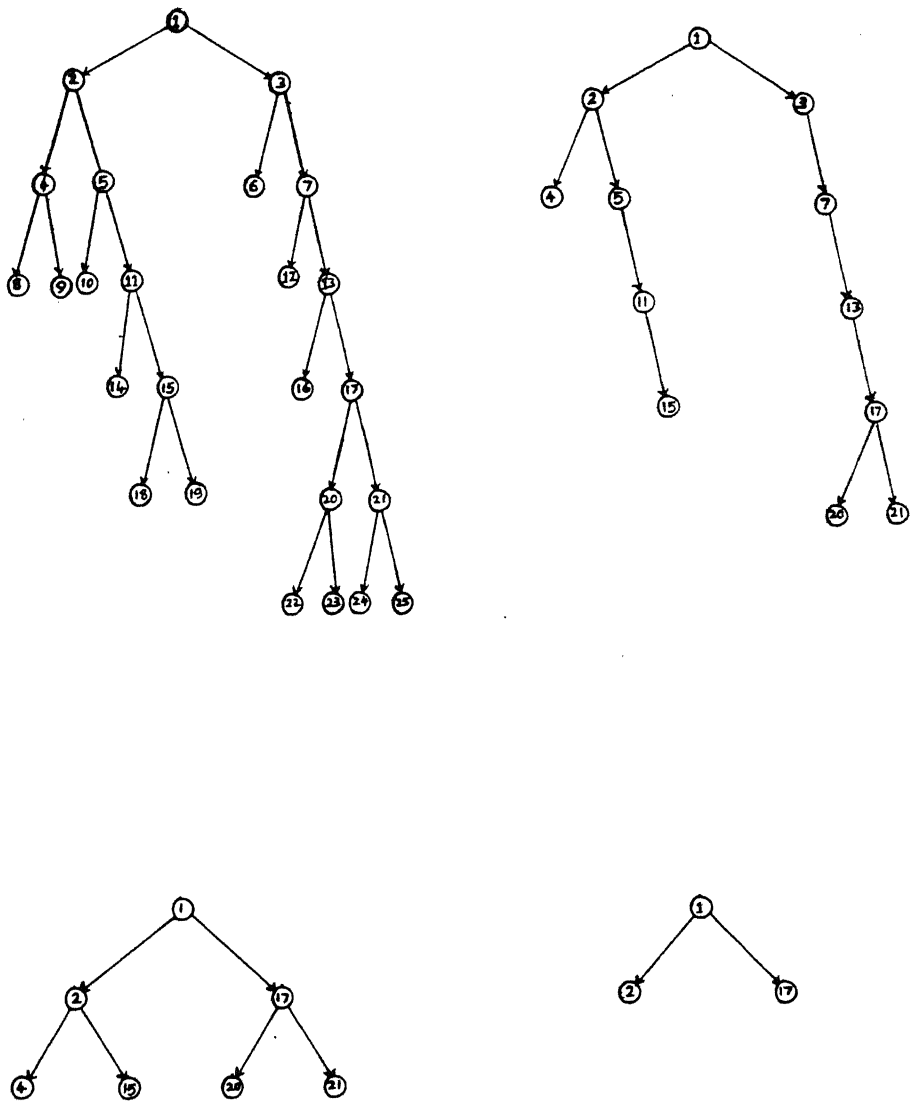


Figure 3.3: An example for Reversed tree method

For each triplet x, y, z = 1,2,...n if edge $y \rightarrow z$ is present is $IG(0)$ then enter the edge $y \rightarrow x$ in $IG(x)$

Step 3 [connect leaf nodes to roots]

For each pair x, y = 1,2,...n

if Level($y \mid T(x,j)$) = 2^j then enter $x \rightarrow y$ in $IG(x)$ along with Dist($y \mid T(x,j)$)

Step 4 [Compress the linear part]

Using the pointer doubling , remove any linear parts in the IG.

step 5 [Node removal]

Calculate the distance of the terminal nodes and remove them

Step 6 [loop]

Repeat Step 5 and 6 for at most $\lceil \log n \rceil$ times.

Step 7 [calculate for every node]

Repeat for $\lceil \log n \rceil$ times.

reconstruct the linear parts and calculate the distance for the node inside the linear parts

Step 8 [building $T(x, j+1)$]

for each pair x, y = 1,2,...n

if y is select in $IG(x)$ then y is connect to $T(x, j+1)$

The intermediate graph is constructed to reduce the number of processors required for the algorithm. Intermediate graph is constructed by taking the edges from all the $T(x,j)$s. One intermediate graph is constructed for each tree $T(x,j)$. Distance of terminal nodes from the root nodes is calculated in each tree $T(x,j)$ and edges are added between the terminal nodes and the root in the termediate graph of the tree $T(x,j)$. Distances of the terminal nodes are also written along wich the edges. The maximum distance of the nodes is

calculated on the intermediate graphs construct. Although the method described here has been proved only for structured graphs , we strongly feel that this can be extended to the general problem as well. The maximum distances on the intermediate graphs are calculate in the steps 4 to 6. This calculation is done in two phases. In the forward phase , the maximum distance is calculate only for some nodes. The the reverse phase of the algorithm , the maximum distance is calculated for the nodes which are eliminated during the first phase. The second phase of the calculation is essentially replay of the forward phase in the reverse order. When the second phase is executed the distance is known for the some of the nodes and the distance of the remaining nodes are calculated based on these values.

Intermediate graph construction can be carried out with $O(n^2)$ processors in constant time. Each tree contains at most n edges and there are only n tree. Hence , the total number of edges which are to considered for the intermediate graph construction is only n^2. Making one copy for each tree $T(x, j)$ take ne processors. The intermediate graph contains at most e edges and we want to make n copies of the intermediate graph. This can be done by simple data propagation routine which uses n processor for each edge in the intermediate graph. This implies that the total number of nodes require for this stage of the algorithm is ne. The distance calculation is carried out in all the intermediate graphs in parallel. The distance calculation technique requires at most e processor for each intermediate graph. Hence , the distances can be calculated using ne processors in $O(\log n)$ time.

Chapter 4

Line graphs

4.1 Introduction

A line graph represent the interrelation between the edges of a graph , root graph. The line graphs are useful to study the properties of the original graphs. A lot of work has been done in the area of eulerian and hamiltonian paths on line and root graphs. The points of a line graph , $L(G)$, represent the edges of the root graph and if any two edges of the root graph G are adjacent then corresponding nodes of the line graph are adjacent. The line graphs can be constructed for directed graphs too. The line graph of a digraph D is the digraph $L(D)$ having a vertex $l(a)$ for each arc a of D and an arc $(l(a_1),l(a_2))$ for each pair of arcs a_1 , a_2 of D which are of the form $a_1 = (u,v)$ and $a_2 = (v,w)$. An input graph H is a line graph if it is isomorphic to the line graph $L(G)$ of G where G is a root graph. The graph G is called the root graph of H where H is a line graph . The line graph detection problem has been studied by many people. Lehot[LE 74]has proposed a sequential algorithms to detect line graph and to construct their root graphs. Except in a trivial case like triangle and star with three branches , root graph of a line graph is unique up to isomorphism. One

obvious way o detect line graphs is to look for forbidden subgraphs 4.1 inside the input graph. Beineke has proved that a graph H is a line graph if only if it does not contain any of the nine forbidden subgraphs. A sequential algorithm can be designed based on this characterization of the line graphs. However , the performance of this algorithm will not be good.

4.2 Algorithms

The algorithms for this problem has two parts. The first part of the algorithm constructs the root graph from the input graph assuming that the input graph is a line graph. The second part of the program generates line graph from the constructed root graph and checks against the input graph. If it differs from the input graph then it indicates that the input graph has at least one of the forbidden subgraph as its induced subgraph. In other words , input graph is not a line graph. If the generated line graph is isomorphic to the input graph then the input graph is a line graph and the generated graph is its root graph.

4.2.1 Labeling algorithm

The sequential algorithm is based on labeling technique. The labeling algorithm is used for detection of line graphs and construction of their root graphs. Each node of the graph is labeled with a pair of numbers. If both the numbers of the node are known , then the node is called fully labeled and if only one number is known the node is called half-named. The numbering of the nodes of input gives a way to map the input graph to the edges of the line graph. The line graph can be generated directly without numbering the nodes. If this method is followed then even if the input graph is a line graph it will be difficult to get a mapping from the nodes of the input graph to edges of the line graph.

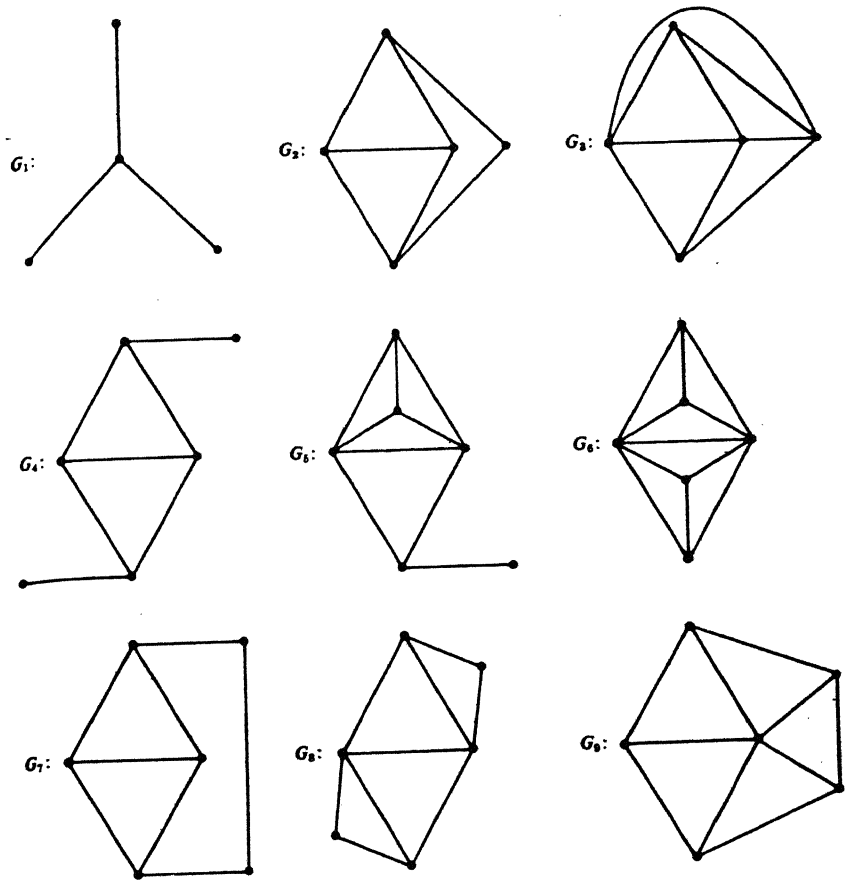


Figure 4.1: Forbidden subgraphs for line graphs

Step 1

Pick up two adjacent nodes. Name them 1-2 and 2-3. They are the two basic nodes to start with.

Step 2

Find all nodes adjacent to both the basic nodes.

Case I [If there is only one such node]

Call it x. If there is a node adjacent to one node of the triangle without being adjacent to any other node of the triangle , then x = 2-4. Otherwise , x = 1-3

Case II [If there are two adjacent nodes]

Call the two adjacent nodes x and y.

1. If x and y are adjacent , there is no cross node and go to step 6.

2. If x and y are not adjacent , they constitute two triangles with the basic nodes, and we find out if there is an odd triangle. If there is one , the corresponding summit , say x , is 2-4 , and then y will be named 1-3 and will be named 1-3 and will be the cross node.

Case III [If there is group of adjacent nodes]

If there is a node a of the group which is not adjacent to a certain node b then either a is the first node of the group under investigation or it is not . In the first case , the tie is broken by examining the adjacency of a to a third node of the group. If it is adjacent , then b is declared the cross node. If it is not , then a is declared the cross node.

Step 3

All the node of the clique are named and cross nodes , if any , is fully named.

Step 4

The nodes of the clique are used to half-name successively the "not-yet-fully-named" nodes which are adjacent. Then the two associated cliques are disposed of , and the naming process takes place as usual.

Step 5

If there is a half-named node adjacent to full named node

Set the half-name node and the fully-named node to basic nodes and go to Step 2.

Step 6

Mark the edges of $L(G)$, the line graph of G , which are in H , until one edge of H is not in $L(G)$. If this happens: Exit-H is not a line graph. If this does not happen: Exit - H is a line graph.

Time complexity of the algorithm is easy to establish. Edges of the line graph are scanned at most twice. A nodes is half-named when it is scanned for the first time and the same node is fully-named when it is scanned for the second time. Constant amount of work is done for half-naming and fully-naming the nodes. Hence , worst case time complexity of the algorithm is $O(e)$.

4.2.2 Parallel algorithm

Each node of root graph is represented by a clique in the line graph. The algorithm recreate the root graph from the line graph by mapping the cliques in the graph to the nodes of the root graph. The algorithm constructs cliques in two phases. At most $2n$ cliques are constructed to detect the root graph. If the input graph is a line graph then the constructed intersection graph is nothing but the root graph of the given line graph. Otherwise , the intersection graph is not a root graph. The cross nodes are possible only in the first set of cliques. The cross nodes are detected by checking for the odd and even triangles is the

nodes of the cliques. Triangle checking can be eliminated by constructing the cliques thrice and the cross nodes can be detected by counting the occurrence of the nodes in the cliques. The algorithm employs $O(n^2)$ processors to detect the line graph and to construct the root graph of the input line graph. The detection algorithm works with $O(\log n)$ time.

Step 1

For each node in the graph select a neighboring input node. i.e n pairs of nodes are selected.

Step 2

For each pair (x, y) find all z such that both x and y are adjacent to z. each group forms a clique in the input graph and corresponds to a node in the line graph.

Step 3

For each clique formed

1. Find the cross node , if any , and remove it from the clique.
2. Order the nodes in the clique in the ascending order and select the smallest three numbers in each group. Number the cliques with their selected nodes

Step 4

Sort the formed cliques by their number and remove duplicate entries.

Step 5

1. Collect the remaining neighbors of the first node of each clique and form new cliques
2. Nodes of each new clique is sorted in the ascending order and the first three numbers used to number the cliques.

Step 6

The intersection graph of the cliques which the root graph is constructed.

Step 7

The line graph is generated from the root graph and checked for isomorphism.

Chapter 5

Concluding remarks

In this thesis three types of problems have been considered. First one is the problem of evaluating a set of expressions. It was shown that the set can be evaluated with less computational cost if there exist some common subexpression. The evaluation method assigns one processor for each unique subexpression in the set. This reduces the processors requirement for the computation. One possible extension to this method is applying node coloring algorithms to identify the a set of non-overlapping nodes.

The tree contraction method evaluates the expression by adjusting the linear expressions stored at various nodes after every rake operation. This ensures that the tree , through out the computation process , represents the same value. Evaluation of the linear expression involves the simplification of expressions which make use of the algebraic properties of the operators. One can attempt to devise a general algorithm which can work with any combination of operators. This can be achieved if one postpones linear expression computation till it is actually required. This can be achieved by storing an expression tree at each of the nodes in the network and computation must be done in the second phase of the algorithm where the subexpressions are computed.

The second type of problem which has been discussed in chapter 3 is the problem of computing the shortest and the longest paths on the directed acyclic network using tree merge algorithm and matrix multiplication. One needs $O(n_3)$ processors for the algorithms which are using matrix multiplication or the normal tree merge algorithm. We have proposed two algorithm which employs less number of processors. First one proposed requires less number of processors. However , we could not prove polylog time bound for this algorithm. The empirical performance of this algorithm in term of the number parallel steps is closer to performance of the $O(n^3)$ algorithms. The second algorithm was designed for a class of directed acyclic graphs. This computes the shortest and longest paths , from a single source , in $O(\log^2 n)$ time using $O(ne)$ processors.

The third problem which is discussed in chapter 4 is the problem of detecting the line graphs and constructing the root graph if the input graph is line graph. An algorithm has been proposed which requires $O(n^2)processorstocomputeinO(\log n)$ time on an CREW machine. The algorithm described for the the undirected graphs can be easily modified for the directed graphs.

REFERENCES

[AH 74] Aho A.V, Hopcroft J.E and Ullman J.D *The design and analysis of computer algorithms* Addison-wesley 1974

[AM 91] Ahuja R.K, Magnanti T.L and Orlin J.B *Network flows: Theory, algorithms and applications* , 1991

[AK 89] Akl S.G *The design and analysis of parallel algorithms* Prentice Hall 1989

[AS 87] Akl S.G and Santoro N *Optimal parallel merging and sorting without memory conflicts* IEEE trans on computers Vol C-36 No 11 1987 pp 1367-1369

[BR 74] Brent R.P *The parallel evaluation of general arithmetic expressions* JACM 21 1974 pp 201-206

[BK 73] Brent R.P Kuck and Maruyama *The parallel evaluation of arithmetic expressions* IEEE Trans. on computers C-22 1973 pp 532-534

[CH 90] Chaudhuri P *An adaptive parallel algorithm for analyzing AOE networks* Operations Research Letter 9 1990 pp 341-34

[CG 86] Chaudhuri P and Ghosh R.K *Parallel algorithm for analyzing activity network* BIT 26 1986 pp 418-429

[CV 86] Cole R and Vishkin U *optimal parallel list ranking* Information and Control Vol 70 1986 pp 35-53

[CO 91] Cormen T.H *Introduction to algorithms* , 1991

[DN 81] Dekel E , Nassimi D, and Sahani S *Parallel matrix and graph algorithms* SIAM J. computing (10) 1981 pp 657-673

[DE 74] Deo N *Graph Theory with applications to engg.* Prentice-Hall, Englewoodcliffs, NY 1974

[DU 65] Duffin R.J *Topology of series-parallel networks* JMAP 10 1965 pp 303-318

[EH 82] Ehab S. El-mallah and charles J. colbourn *Optimum communication spanning trees in serial-parallel networks* 1982

[GM 84] Gazit H and Miller G.L *A parallel algorithm for BFS of a directed graph* IPL (19) 1990 pp 678-704

[GB 84] Ghosh R.K and Bhattacharjee G.P *Parallel breadth first algorithm for trees and graphs* IJCM (15) 1984 pp 255-268

[GB 86] Ghosh R.K and Bhattacharjee G.P *Parallel shortest path algorithm* IEEE proceedings (133) 1986 pp 87-93

[HA 69] Harary F *Graph theory* Addison-Wesley 1969

[HS 84] Horowitz E and Sahani S *Fundamentals of computer algorithms* Galgotia publications , 1984

[KN 81] Knuth D.E. *Art of computer programming vol 1* Addison-Wesley , 1981

[KD 88] Kosaraju S.R and Delcher A.L *Optimal parallel evaluation by raking* Extended Abstract , 1988.

[KM 75] Kuck D.J and Maruyama K *Time bounds on the parallel evaluation of expressions* SIAM J. computing vol 4 no 2 1975 pp 147-162

[LE 74] Lehot P.G.H *An optimal algorithm to detect a line graph* JACM 21 1974 pp 569-575

[MR 86] Miller and Reif On optimal evaluation of expression 26thSymposium on foundations of computer science , pp 478-489 , 1985

[PM 75] Preparata F P and Muller D.E *The time required to evaluate arithmetic expressions* IPL 1975 Vol 3 No 5 pp 144-146

[QU 88] Quinn M.J *Designing efficient algorithms for parallel computers* Mcgraw Hill 1988

[SI 90] Siegal H J *Interconnection networks for large scale parallel* processing Mcgraw-Hill , 1990

[ST 69] Strassen , v. *Gaussian elimination is not optimal* Numerische mathematik 1969 vol 13 pp 354-356

[SY 82] Syslo M. M. *A labeling algorithm to recognize a line digraph* IPL Vol 15 no 1 1982 pp 28-30

[TN 82] Takamizawa K, Nishizeki T, and Saito N *Linear time computability of combinatorial problems on serial parallel graphs* JACM 29 1982 pp 623-641

[WI 75] Winograd S *On the parallel evaluation of certain arithmetic expressions* JACM vol 22 , no 4 1975 pp 477-492